Deep Learning Project 2
Automatic Differentiation

Etienne Salimbeni, Matthias Zeller, Fatih Mutlu
EPFL, Switzerland

***Abstract*—Computing derivatives is a major task in Machine Learning, in order to perform gradient-based (or higher order) optimization on an objective function. In this work, we present the implementation of our backpropagation framework, build on top of PyTorch's tensors, without using `torch.autograd`.**

I. Introduction

Three main methods exist to compute derivatives: *Numerical Differentiation*, *Symbolic Differentiation* and *Automatic Differentiation* (AD, also called Algorithmic differentiation) [1], [2]. Only symbolic differentiation and AD compute exact derivatives, up to floating point precision. The former generates symbolic expressions, which can become quit complex for long composite functions as encountered in deep learning. On the other hand, AD performs elementary operations directly on local gradients that propagate through the computational graph, and thus became a mainstream tool in many machine learning frameworks [1].

Let us recall the basics of multivariable calculus. Assume we have a function $f : \mathbb{R}^n \to \mathbb{R}^m$ composed of L functions:

$$f(\mathbf{x}) := f^{(L)}(f^{(L-1)}(\dots f^{(2)}(f^{(1)}(\mathbf{x}))\dots))$$

The Jacobian of f is

$$J_f := \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix},$$

and can be computed with the matrix product of the Jacobians:

$$J_f = J_{f^{(L)}} \cdot J_{f^{(L-1)}} \dots J_{f^{(2)}} \cdot J_{f^{(1)}} .$$

There are (mainly) two different versions of automatic differentiation (AD): *forward mode* and *reverse mode*. The former computes derivatives of the output $f(\mathbf{x})$ with respect to one input x_j and effectively computes one column of the Jacobian J_f, whereas reverse mode AD handles adjoint variables, that is derivatives of one output component f_i with respect to the input $\mathbf{x}$, and effectively computes one row of the Jacobian J_f [2]. In the context of *deep learning*, the input is high dimensional whereas output is low dimensional (i.e., $m \ll n$), which makes the forward mode computationally intractable. Actually, in most applications, the objective function is one-dimensional ($m = 1$), and the full gradient can be computed in one reverse mode sweep. However, note that reverse mode is intrinsically memory inefficient: we must first perform the forward pass, while bookkeeping all intermediate computations since they are needed for the backward pass.

II. Implementation

Our framework is split into two main classes: `Tensors` and `Modules` (which we will call operators). Tensors wrap `torch.tensor` to store actual data, its gradient, and other components used to compute the gradient. Each tensor is a *node* in the computational graph, whereas operators *manipulate* the graph (e.g. create new node as results of an operation and prepare the backward pass accordingly).

`Tensors` have two elements used to compute gradients:

- `parents`: a list of `Tensors` that created the current tensor, this allows to walk backwards through the graph during the backward pass
- `backward_fun`: a lambda function (assigned by operators) which takes as parameters the gradient with respect to the output, and the inputs of the operator that created the current tensor.

`Tensors` overload all elementary operations: addition, subtraction, multiplication (Hadamard product), matrix multiplication, sum of all elements, and transposition. In turn, `Tensor` calls the corresponding operators (subclasses of `Module`) by passing a reference to itself and other inputs (if any). For instance, addition is handled as:

```
class Tensor:
    def __add__(self, other):
        return F.Add()(self, other)
```

There are two types of operators, *elementary* and *composite* operators. They effectively differ by the way they manipulate the computational graph. This difference is handled by the inheritance relationship between the superclass `Module` and the subclasses, and depends on which methods are reimplemented. There are two public methods `forward` and `backward` and two protected methods `_forward` and `_backward`. The superclass `Module` implements:

- `forward(*inputs)`: first computes the actual result by calling `_forward(*inputs)`, then assign the `Tensor`'s `backward_fun` to the operator's `backward` function. Also, set the `parents` of the result tensor to the list of input tensors. Eventually return the result.
- `backward(output, *inputs)`: call the protected `_backward` method, modifies the gradient of input tensors in place and returns nothing.

Thus, `Module` lets the protected methods `_forward` and `_backward` as abstract methods. Elementary operators implement those two protected methods, i.e. they implement only the computations that are unique to the operation of interest, and they let the superclass manage computations that are common to all elementary operations.

On the other hand, composite operators have no backward function, and re-implement only `forward`: they don't directly manipulate the computational graph, but rather manipulate data in terms of elementary operations.

This framework is analogous to PyTorch: a user can create custom modules by implementing the `forward` function in terms of other functions provided by the API.

A. Backward pass

We must introduce the notion of an adjoint variable. Each node x_i in the computational graph has an associated adjoint variable $\bar{x}_i := \frac{\partial y_k}{\partial x_i}$, with y_k the k-th output. In our case, the output is a scalar, which we call the loss $\mathcal{L}$. For the optimization procedure used to train a network, what we call the gradient is actually the adjoint. During the forward pass, the value of intermediate nodes x_i are computed and retained in memory [1]. The backward pass is performed when the user calls `loss.backward()`:

```
class Tensor:
    def backward(self):
        self.grad = torch.tensor([[1.0]])
        self.backward_fun()
        # walk backwards through graph...
```

That is, we first initialize $\bar{\mathcal{L}} = 1$, call its `backward_fun()` to accumulate the gradient in its parent nodes, and then walk backward through the graph to call each node's `backward_fun()`. This effectively computes all adjoint variables $\bar{x}_i$. Those functions `Tensor.backward_fun` are actually the functions `Module.backward` of the operators that created the tensor. As mentioned above, the actual gradient accumulation is implemented in the protected functions `_backward`, which are all of the form:

```
def _backward(self, output, *inputs):
    inputs[0].grad += ...
    inputs[1].grad += ...
    ...
```

Note that we *add* something to the inputs' gradient, instead of setting a new value. Indeed, the gradient of a node is a sum of the contribution of all its children node in the computational graph. In the scalar case, each adjoint is computed as:

$$\bar{x}_i = \sum_{j:\, x_j \text{ child of } x_i} \bar{x}_j \frac{\partial x_j}{\partial x_i},$$

[1]Note that the storage of intermediate values is handled by the Python garbage collector: the `backward_fun` of `Tensors` are lambda functions which reference the results of an operation. This is probably less efficient but less error prone than manually handling memory.

where $\bar{x}_j$ is `output.grad`, and $\frac{\partial x_j}{\partial x_i}$ is encoded in the body of `_backward`.

B. Activation functions

We implemented the following activation functions, for which we specify the backward computations:

- $\text{relu} : \mathbf{x} \mapsto \mathbf{y}, \quad \bar{\mathbf{x}} = \bar{\mathbf{y}} \odot \mathbb{I}\{\mathbf{x} > \mathbf{0}\}$
- $\tanh : \mathbf{x} \mapsto \mathbf{y}, \quad \bar{\mathbf{x}} = \bar{\mathbf{y}} \odot (\mathbf{1} - \mathbf{y} \odot \mathbf{y})$
- $\text{sigmoid} : \mathbf{x} \mapsto \mathbf{y}, \quad \bar{\mathbf{x}} = \bar{\mathbf{y}} \odot \mathbf{y} \odot (\mathbf{1} - \mathbf{y})$

where $\odot$ is the Hadamard product, and $\mathbb{I}$ the indicator function, which together with $>$ is applied component-wise. For tanh and sigmoid, we take advantage of the fact that $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ can be expressed in terms of the output $\mathbf{y}$. For instance, in the scalar case:

$$y = \tanh(x) \to \frac{dy}{dx} = 1 - \tanh^2(x) = 1 - y^2$$
$$y = \text{sig}(x) \to \frac{dy}{dx} = \text{sig}(x) \cdot (1 - \text{sig}(x)) = y \cdot (1 - y)$$

C. Layers

A `Layer` is a subclass of `Module`, with two additional attributes: the input and the output dimension,

```
class Layer(Module):
    def __init__(self, n_in: int, n_out: int):
        super(Layer, self).__init__()
        self.n_in = n_in
        self.n_out = n_out
```

With this framework, implementing a linear layer (a composite operator) is as simple as defining the forward pass in terms of matrix-vector operations:

```
class LinearLayer(Layer):
    def forward(self, x):
        return self.W @ x + self.b
```

where the omitted body of `__init__` performs parameter initialization, either with the standard normal distribution, or with xavier initialization if `xavier_init=True`. The `Sequential` module is straightforward as well: we simply sequentially feed the ouptut of each layer to the input of the next layer:

```
class Sequential(Layer):
    def __init__(self, *layers):
        self.layers = layers
        n_in = self.layers[0].n_in
        n_out = self.layers[-1].n_out

        super(Sequential, self).__init__(n_in, n_out)

    def forward(self, x: Tensor):
        for l in self.layers:
            x = l(x)
        return x
```

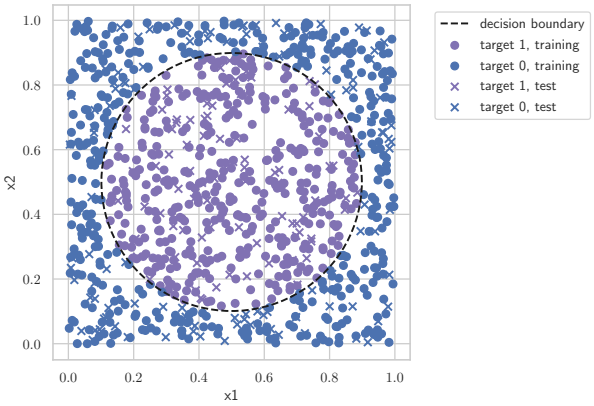

Fig. 1. An instance of the generated random dataset, with 10,000 samples, split into training and test set with 80%-20% ratio.

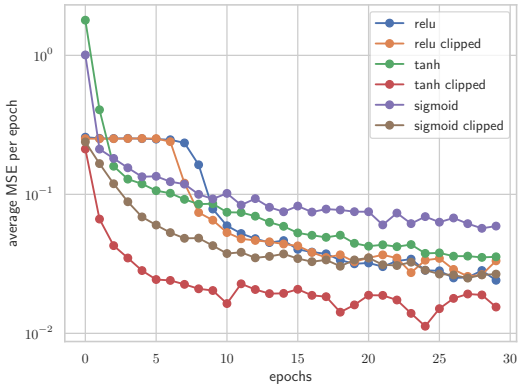

Fig. 2. Typical learning curves obtained for the six models. *Clipped* refers to applying sigmoid to the 1D output. Model-specific learning rates are used, selected by 5-fold cross validation. Each dot is the average MSE per epoch. Samples are shuffled at each epoch.

III. Results

We test and compare three different models implemented with our framework. To do so, we generate a dataset of $N = 10,000$ samples in 2D, where each feature x_i, $i = 1, 2$ is uniformly distributed within $[0, 1]$. The label of a sample $l(\mathbf{x})$ is defined as:

$$l(\mathbf{x}) := \begin{cases} 1 & \text{if } (x_1 - 0.5)^2 + (x_2 - 0.5)^2 > \frac{1}{2\pi} \\ 0 & \text{otherwise} \end{cases}$$

A typical instance of this dataset is depicted in Figure 1.

The network architecture is fixed: 3 hidden layers of 25 neurons each, 2 input units and 1 output unit. We tested *six different models*, using one of the three activation functions (`Sigmoid`, `ReLU`, `tanh`) between the hidden layers, and either a linear output or a "clipped" output (passing the 1D output through `Sigmoid`). The clipping is motivated by the fact that `MSELoss` with linear output will also penalize samples that are correctly classified. `Sigmoid` will map the output to $[0, 1]$, which attenuates the problem.

We first performed some hyperparameter tuning to find the best learning rate of each model: running the `test.py` script with the `--cv` argument performs grid search on the learning rate, using 5-fold cross validation implemented by the function `kfold_cv` in our `training` module.

A typical training procedure is depicted in Figure 2, as obtained by running `test.py` without arguments. Performance evaluation is assessed by running `test.py --stats`, which runs 5-fold cross validation two times on each model, results are reported in Table I. The clipped tanh model performs the best, followed by the sigmoid and tanh models. The relu and clipped relu models perform the worst, probably due to the initial saturation phase (visible in Figure 2).

Model	Training accuracy	Test accuracy
tanh	0.983 ± 0.00419	0.969 ± 0.00811
tanh clipped	0.985 ± 0.00560	0.970 ± 0.00846
relu	0.976 ± 0.0135	0.965 ± 0.0249
relu clipped	0.968 ± 0.0125	0.964 ± 0.0141
sigmoid	0.980 ± 0.00581	0.970 ± 0.0145
sigmoid clipped	0.970 ± 0.0110	0.960 ± 0.0124

TABLE I
Training and test accuracies of each model, computed as mean ± std. Values are computed on the 2x 5 folds (5-fold CV run two times).

IV. Conclusion

We were able to set up a framework that implements automatic backpropagation and was shown to work on a simple dataset. As our framework is built on top of PyTorch but fully implemented in Python, it lacks computational efficiency. Indeed, although actual tensor operations are performed by Pytorch, and thus are optimized, the computational graph and the data flow is fully handled in Python. Moreover, our framework is not parallelizable as-is, and only one sample can be fed through a network at once. Finally, many computational optimizations exist for automatic differentiation [3], but are not explored here.

Despite these limitations, our framework exhibits a great flexibility, and makes it easy to add any new activation function or module. Also, its usage is almost identical to PyTorch's, which makes it easy and intuitive to use. Furthermore, this project provides some pedagogical value: inspection of code written in a high-level language enables understanding what happens under the hood when using libraries like PyTorch (if complex optimization procedures are put aside).

References

[1] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *arXiv:1502.05767 [cs, stat]*, Feb. 2018, arXiv: 1502.05767. [Online]. Available: http://arxiv.org/abs/1502.05767

[2] C. C. Margossian, "A Review of automatic differentiation and its efficient implementation," *WIREs Data Mining and Knowledge Discovery*, vol. 9, no. 4, Jul. 2019, arXiv: 1811.05031. [Online]. Available: http://arxiv.org/abs/1811.05031

[3] D.-I. A. Kowarz, "Advanced Concepts for Automatic Differentiation based on Operator Overloading," Mar. 2008.